Sparse matrix multiplication: The distributed block-compressed sparse row library

Borštnik, Urban ; Vande Vondele, Joost ; Weber, Valéry ; Hutter, Jürg

DOI: https://doi.org/10.1016/j.parco.2014.03.012

Sparse Matrix Multiplication: The Distributed Block-Compressed Sparse Row Library

Urban Borštnik[a,1,*], Joost VandeVondele[b], Valéry Weber[a,2], Jürg Hutter[a]

[a] *Physical Chemistry Institute, University of Zurich, Winterthurerstrasse 190, CH-8057 Zürich*
[b] *Department of Materials, ETH Zurich, Wolfgang-Pauli-Strasse 27, 8093 Zurich, Switzerland*

Abstract

Efficient parallel multiplication of sparse matrices is key to enabling many large-scale calculations. This article presents the DBCSR (Distributed Block Compressed Sparse Row) library for scalable sparse matrix-matrix multiplication and its use in the CP2K program for linear-scaling quantum-chemical calculations. The library combines several approaches to implement sparse matrix multiplication in a way that performs well and is demonstrably scalable. Parallel communication has well-defined limits. Data volume decreases with $\mathcal{O}(1/\sqrt{P})$ with increasing process counts P and every process communicates with at most $\mathcal{O}(\sqrt{P})$ others. Local sparse matrix multiplication is handled efficiently using a combination of techniques: blocking elements together in an application-relevant way, an autotuning library for small matrix multiplications, cache-oblivious recursive multiplication, and multithreading. Additionally, on-the-fly filtering not only increases sparsity but also avoids performing calculations that fall below the filtering threshold. We demonstrate and analyze the performance of the DBCSR library and its various scaling behaviors.

Keywords: Sparse Matrix, Parallel Sparse Matrix Multiplication, Quantum Chemistry

1. Introduction

Dense matrix-matrix multiplication is one of the most basic operations in linear algebra. Highly optimized implementations, both serial and parallel, are available and the underlying algorithms are well understood [1, 2, 3, 4, 5, 6].

*Corresponding author. Email: urban.borstnik@id.ethz.ch
[1] While on leave from the National Institute of Chemistry, Ljubljana, Slovenia. Present address: High Performance Computing Group, Information Technology Services, ETH Zurich, SOW H 12, Sonneggstrasse 63, CH-8092 Zurich, Switzerland
[2] Present address: IBM Research Division, Zurich Research Laboratory, 8803 Ruschlikon, Switzerland

Somewhat surprisingly, the same is not true for sparse matrix-matrix multiplication [7, 8, 9, 10, 11, 12, 13]. The reason might be that many problems can be solved based on a sparse matrix vector multiplication kernel. In the field of computational chemistry, physics, and material science an important exception can be found, namely the problem of solving the self consistent field (SCF) equations that arise in Kohn-Sham or Hartree-Fock theory. The solution of the SCF equations is a matrix that minimizes an energy functional subject to constraints, such that the solution matrix is idempotent ($\mathbf{A} \times \mathbf{A} = \mathbf{A}$). Traditionally this solution matrix, named the density matrix, is found using diagonalization techniques. Typically 10-50% of the eigenvectors of the Hamiltonian matrix of the system are needed to build the density matrix. As the system size increases, both the density matrix and the Hamiltonian matrix become sparse but the eigenvectors do not. This is an opportunity to avoid the cubically scaling diagonalization step and to directly compute the density matrix using linear scaling techniques in which the computational effort scales linearly with an increase in system size. Among the various options, the density matrix can be obtained as a (matrix) function of the Hamiltonian matrix. The Hamiltonian matrix can itself be obtained as a Chebyshev expansion or from recursion relations [12]. In both cases the most important operation is sparse matrix-matrix multiplication. As a result, several groups that develop linear scaling SCF methods have reported on the development of sparse matrix-matrix multiplication algorithms and libraries.[7, 10, 11, 12, 13, 14]

The sparsity of the Hamiltonian and density matrix depends on the chemistry of the underlying system, the geometric arrangement of the atoms, and the choice of the basis set. For the common case of atom-centered basis sets, the magnitude of the matrix elements decays with increasing distance between the atoms. Typically, for a three dimensional atomic system, 10'000s of matrix elements per row are non-negligible compared to a given threshold. Product matrices retain this sparsity ratio. The computational cost for multiplication is therefore large, and parallel computing is thus essential to have a reasonable time to solution. For a wide range of interesting problems that are affordable on current supercomputer hardware, the occupation is high (i.e., the percentage of non-zero elements) is thus large, on the order of 10%, but remains so during the procession of multiplications in the calculations. Good performance in the limit of such high occupations should therefore be an important design criterion. A further aspect that has to be considered is the fact that some internal structure is present in the sparsity pattern. In particular, matrix elements are naturally grouped into "atomic blocks" or sub-matrices, which correspond to the interactions between basis functions centered on a given pair of atoms. It is natural and efficient to use this structure to enhance the performance of the implementation.

We present a sparse matrix multiplication library which takes these two points into account and aims to provide good performance for these types of matrices. In particular, we aim for an algorithm that becomes equal to the known optimal algorithms for the dense matrix multiplication in the case of a sparse matrix with 100% occupation. In particular, all-to-all communication

is avoided in that case, too. Furthermore, a cache-oblivious strategy is introduced that enhances the FLOP rate, and an autotuned library is developed that performs small matrix multiplications efficiently. Because the library is used not just for matrix multiplication but also other matrix operations as well as a storage container for matrices, the design choices are constrained by these requirements.

In the following section we present the implementation of the DBCSR sparse matrix multiplication library. We describe the data storage layout used, data distribution and transfer among distributed-memory nodes, the node-local multiplication approach. In the Results and Discussion section we present the performance of the library for common sparsities and core counts. The performance of an actual application is analyzed in the Application Performance: Linear Scaling Computational Cost section. This section is followed by the Performance Limits in which the performance at the strong and weak scaling limits is reported and analyzed.

2. Sparse Matrix Multiplication

2.1. Matrix Storage

DBCSR (Distributed Blocked Compressed Sparse Row) matrices are stored in blocked compressed sparse row (CSR) format distributed over a two-dimensional grid of processes.

Individual matrix elements are grouped into blocks by rows and columns. Block sizes are chemically motivated, e.g., based on the number of basis functions used for an atom type. The blocked rows and columns form a grid of blocks. It is the blocks that are indexed by the CSR index. Indexing the blocks instead of individual elements makes the index smaller since there are far fewer blocks than individual elements in a matrix for most basis sets. Indexing by blocks also makes lookups by atom number, which is a very common operation, much easier.

The blocks of the matrix are distributed over a rectangular process grid. While an arbitrary rectangular grid can be used, the dimensions of the process grid are chosen so that the least common multiple of the two grid dimensions is minimized, as we later explain. Square grids are preferred because they minimize the number of messages exchanged. The blocked matrix rows and columns can be arbitrarily mapped to the process rows and columns. Two mapping functions are defined: $p_r(r)$ maps block rows r to process rows $p_r(r)$ and $p_c(c)$ maps block columns c to process columns $p_c(c)$. Any matrix block (r, c) is therefore assigned to a process $(p_r(r), p_c(c))$ from the process grid. To prevent load imbalance it is best to provide a mapping in which rows of equal block size are evenly distributed among the process rows and columns. In practice, the randomization is performed once at the beginning of a program and is used for all subsequent matrix operations, including matrix multiplication. A sample permutation of a diagonal-heavy sparse matrix is shown in Fig. 1. In addition it is beneficial for the row and column mappings to be similar. The library

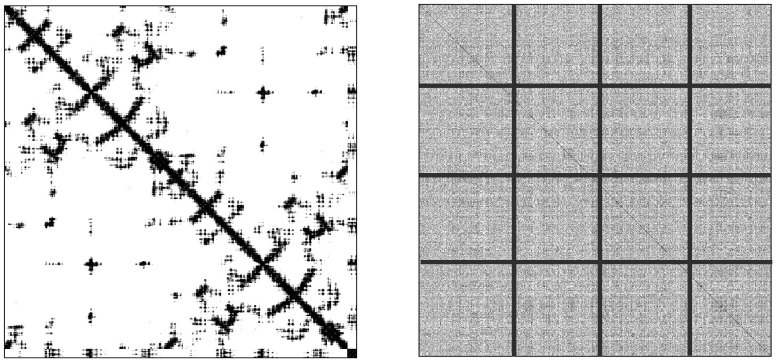

Figure 1: Randomization of a sparse matrix. An unmodified matrix is shown in subfigure a. The same matrix after randomization is shown in subfigure b with an overlayed 4×4 16-process grid. The more uniform density after randomization helps to achieve good load balancing.

provides a routine to help match the row and column mappings for non-square process grids.

Symmetric matrices, including antisymmetric, Hermitian, and antihermitian matrices, are stored with just one half of the matrix. In these matrices only the upper triangle is stored, i.e., blocks $(r, c), c \geq r$. To keep data balanced among the processes, these blocks are not all stored on the process row and column corresponding directly to the row and column distribution $(p_r(r), p_c(c))$. Such a mapping would be grossly imbalanced. Instead blocks for which $r + c$ is divisible by 2 are stored at the transposed process in process row and column $(p_r(c), p_c(r))$.

2.2. Matrix Multiplication

Matrix multiplication in the DBCSR library is a combination of local multiplication and data communication. Cache-oblivious sparse matrix multiplication, used in a dense matrix context in Ref. [6], together with autotuned multiplication kernels is used for local multiplication. The inter-node data communication is based on Cannon's algorithm [15]. The general communication scheme for specific cases is similar to work presented in [8] and [16].

2.2.1. Data Exchange for Matrix Multiplication

The communication pattern used for transferring data for the multiplication is based on Cannon's algorithm [15] that is generalized to arbitrary 2-dimensional process grid coordinates $R \times C$ with R process rows and C process columns. While Cannon's parallel matrix multiplication algorithm is based on a square process grid, we introduce a modified scheme in which all matrices are distributed on a $R \times C$ process grid. An additional virtual process dimension, $V = \mathrm{lcm}(R, C)$ is defined to match matrices in multiplication. For a

$\mathbf{C} := \mathbf{C} + \mathbf{A} \times \mathbf{B}$ matrix multiplication, matrix $\mathbf{A}$ is mapped to a $R \times V$ virtual process grid and matrix $\mathbf{B}$ is mapped to a $V \times C$ process grid. Matrix $\mathbf{A}$ is split so that each process column has V/C submatrices while the rows remain unchanged. Similarly, matrix $\mathbf{B}$ is split so that each process row has V/R submatrices while the columns remains unchanged. Accordingly, matrix $\mathbf{C}$ remains unchanged. The steps of the modified Cannon's algorithm proceed according to the virtual dimension V. Every physical process thus emulates one or more virtual processes.

Cannon's algorithm first performs an alignment in which the matrix $\mathbf{A}$ data in process row r is shifted r columns right and matrix $\mathbf{B}$ data in process column c is shifted c rows down. The result of the alignment is that the data needed for the first step of the multiplication is present on each process. After that all communication is only with direct neighbors in the process grid. In our library the alignment is combined with other pre-processing steps, such as matching process rows and columns for non-square process grids, explicitly duplicating the upper and lower half of symmetric matrices and transposing the blocks as needed, and preparing the index for multiplication.

A process (r, c) from the $R \times C$ grid performs the node-local multiplications and additions for the $\mathbf{C}(r, c)$ submatrix of the $\mathbf{C}$ matrix that is local to it according to the distribution. A series of $V = \text{lcm}(R, C)$ matrix $\mathbf{A}$ row shifts and matrix $\mathbf{B}$ column shifts are performed according to Cannon's algorithm so that any process (r, c) obtains and processes all corresponding $\mathbf{A}(r, k)$ and $\mathbf{B}(k, c)$, $1 \leq k \leq V$, submatrices. The matrix $\mathbf{C}$ data always remains resident on its original process. Asynchronous MPI operations are used to transfer the data among the neighboring processes, thus potentially overlapping communication and computation. Because $V = \text{lcm}(R, C)$ steps are performed for $R \cdot C$ processes, the number of steps is minimal when $R = C$ or $V = \text{lcm}(R, C) = R = \mathcal{O}(\sqrt{P})$. In practical terms a square number of processes is optimal, or at least when R and C have most of their factors in common.

A number of approaches were tried prior to settling on the modified Cannon scheme described above. Among them were SRUMMA RMA [5] and a hybrid between Cannon's algorithm and SRUMMA. Ultimately only the modified Cannon's algorithm proved to consistently perform well on all systems we encountered.

A benefit of using a Cannon-based algorithm for communication is its predictable scaling limits. There are always $\sqrt{P}$ steps in the multiplication. Very little additional memory is needed during multiplication. Data volume V decreases as $\mathcal{O}(1/\sqrt{P})$ with increasing process count P. In a bandwidth-limited interconnect with a bandwidth BW, the communication time is $t = V/BW$ and therefore $t = \mathcal{O}(1/(BW\sqrt{P}))$; however, on many interconnects the available bandwidth actually decreases with increasing processes counteracting the inverse scaling with increasing process counts.

Following the row- and column-wise communication pattern in the multiplication, the library creates multiple MPI subcommunicators. Separate subcommunicators are created for each process row and for each process column. Due to problems encountered with many short-lived subcommunicators on some

clusters, the use of subcommunicators can be turned off by the user.

2.2.2. Node-local Multiplication

A local multiplication is performed at every step of the Cannon's algorithm. A submatrix of matrix **A** is multiplied with a submatrix of matrix **B** matrix and the result is added to the local **C** submatrix. The local multiplication is performed in each of the $V = \mathcal{O}(\sqrt{P})$ steps of the modified Cannon's algorithm. Because the local multiplication must deal with blocked sparse matrices with block sizes that are commonly unfriendly to common CPU optimizations, several techniques have been combined to obtain improved performance.

The multiplication is performed in a cache-oblivious manner. The indices of all three matrices are first converted from the CSR format to a sequential index. The indices of matrices **A**, **B**, and **C** are recursively split vertically or horizontally depending on the largest dimension at the current recursion depth [6]. The recursion continues until both **A** and **B** have at most a predefined number of blocks, which is 512 by default. After the recursion limit is reached the **A** and **B** indices are converted to the CSR format to be multiplied using standard CSR sparse multiplication on the level of blocked elements instead of elements themselves.

The CSR multiplication is split into two parts: the first is scanning the index and determining what to multiply and the other is performing the multiplications. The two outermost loops run over matrix **A** rows and columns and the inner most loop runs over columns of matrices **B** and **C**. Rather than directly computing the blocked product and sum $\mathbf{C}(i,j) = \mathbf{C}(i,j) + \mathbf{A}(i,k) \times \mathbf{B}(k,j)$ in the inner loop, parameters of that blocked multiplication are saved to a stack. Seven parameters are in each stack element. They describe the dimensions of the blocks and the location of the data blocks. The stack is processed (i.e., the multiplications performed) when it is full or when the local multiplication is finished. The default stack size is 1000 but is configurable at runtime. Several stacks are filled at once. In addition some stacks are homogeneous, containing only multiplications of the same type; for example all stack entries are multiplications of 5×13 and 13×5 blocks. Decoupling the index generation from the calculations themselves allows better cache reuse and offloading calculations to accelerator devices such as GPUs.

We have written a CUDA implementation of stack processing for GPUs. The CPU always performs all the relatively complex index building, filling the stacks. The stacks are then transferred to the GPU where a CUDA kernel performs all the calculations. Whenever the CPU has no indexing work to do it also processes stacks. For performance reasons, the aim is for the CPU to process stacks with mixed block sizes or smaller block sizes and the GPU processes homogeneous stacks with larger block sizes.

The block sizes in most matrices are based on the number of basis functions used, which are often not common CPU-friendly vector sizes such as 4 or 8. For example, Hydrogen and Oxygen give rise to block dimensions of 5 and 13 when described by the DZVP basis set [17], resulting in 5×5, 5×13, 13×5, and 13×13 blocks. To overcome this, a special automated self-optimizing library,

libsmm, has been developed for multiplying small blocks while ensuring that the performance of a provided BLAS library is always achieved.

SMM Library. The library for small matrix-matrix multiplication (libsmm) aims to get best possible performance for the primitive operation $\mathbf{C} = \mathbf{C}+\mathbf{A}\times\mathbf{B}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are small matrices with selected dimensions. For our applications we typically need matrix dimensions from the set {1, 4, 5, 6, 9, 13, 16, 17, 22, 23}.

Given the n_s selected small dimensions ($n_s = 10$ for the example above), there are ${n_s}^3$ possible combinations for the multiplication of the block sizes with the n_s possible dimensions. A user-level routine calls specialized code for these ${n_s}^3$ cases and calls the BLAS xGEMM routine otherwise. Libsmm is practical for $n_s \approx 10$, given the $\mathcal{O}({n_s}^3)$ specializations needed.

Libsmm can be configured to yield code for real and complex data, single and double precision, and the various combinations of transposition of $\mathbf{A}$ and $\mathbf{B}$. For efficiency reasons, the interface is more restricted than that of the usual DGEMM call: scaling and transposition information is fixed.

Library generation is a three stage processes and is based on automatic code generation with runtime benchmarking.

In the first stage, code for the multiplication of *tiny* matrices is generated, using a list of user provided dimensions that are usually selected in the range of 1–12. For all possible combinations of these tiny dimensions, explicit Fortran code is generated that explores all possible loop orderings and all possible unroll depths explicitly. Note that the use of Fortran code implies portability but might be less effective than generated assembly code. All these variants are benchmarked in a tight loop with $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ being in cache. The fastest version is retained.

In the second stage, code for *small* matrix dimensions is generated in various different ways including a Fortran loop nest, a MATMUL call, a BLAS call, and four different block recursive multiplication variants with block sizes selected from the best performing tiny multiplications. For the final multiplications in these block recursive schemes, the optimal *tiny* code from the first step is used. The best performing code is selected based on the runtime benchmarking.

In the final step, all optimal code variants and the corresponding user level routine are compiled into a library and verified both for correctness and performance.

The performance gains of using the SMM library compared to the MKL library on a Sandy Bridge architecture are shown in Fig. 2.

Filtering. When filtering of the $\mathbf{C}$ product matrix is requested, all blocks $\mathbf{C}(i,j)$, where $||\mathbf{C}(i,j)|| < \epsilon$, are deleted after the multiplication is performed. In addition, on-the-fly filtering can be performed. If on-the-fly filtering is used, block multiplications whose contribution to the final block norm is below a threshold are skipped. Due to its speed the Frobenius norm is used by default. The threshold is defined so that the maximum accumulated error is at most ϵ. The effect is similar to product-space block filtering described in [18]. For each blocked

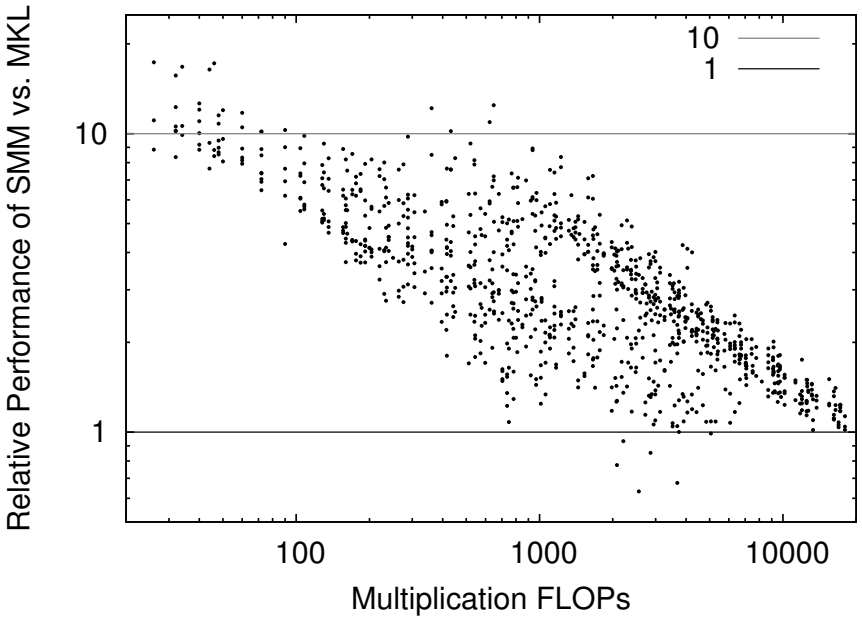

Figure 2: Relative performance for the SMM library compared to Intel MKL for all combinations of matrix dimensions from {1, 4, 5, 6, 9, 13, 16, 17, 22, 23}. The y-axis is the ratio of the time as obtained with SMM vs. the time obtained with MKL. The x-axis is the approximate number of FLOPs needed for the multiplication. All calculations ran on a Sandy Bridge architecture.

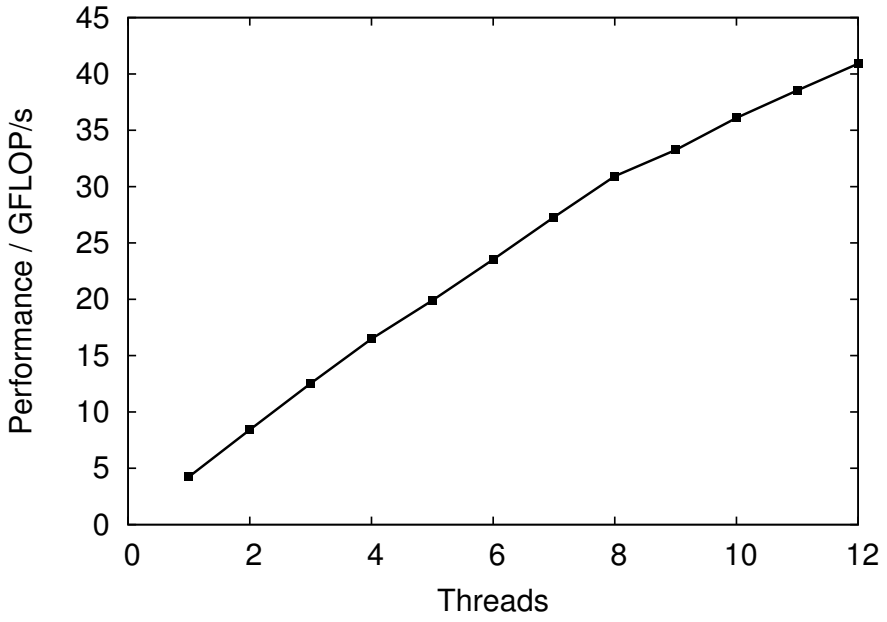

Figure 3: Threading strong scaling performance. The actual total performance for a sample matrix multiplication when using from 1 to 12 threads is presented.

row i of the matrix $\mathbf{A}$, ϵ is divided by n_r, the number of blocks in that blocked row of the matrix, to yield ϵ_i. The number of blocks in a row i of the $\mathbf{A}$ matrix determines the maximum number of possible block multiplications for any element in a row of $\mathbf{C}$. A block multiplication $\mathbf{A}(i,k) \times \mathbf{B}(k,j)$ is performed only if $||\mathbf{A}(i,k)|| \cdot ||\mathbf{B}(k,j)|| \geq \epsilon_i$; otherwise it is skipped. Therefore the difference between the calculated and the exact block will be at most $n_r \cdot \epsilon_i = \epsilon$. After accounting for the additional final filtering, the final calculated block $\mathbf{C}(i,j)$ is within 2ϵ of the exact $\mathbf{C}(i,j) = \sum_k \mathbf{A}(i,k) \times \mathbf{B}(k,j)$. On-the-fly filtering can provide speedups of up to 300% on realistic matrices [19].

For a multiplication of $\mathbf{C} := \mathbf{C} + \mathbf{A} \times \mathbf{B}$ the existing sparsity of the $\mathbf{C}$ product matrix can be enforced during multiplication. This can greatly reduce the computation time when the desired sparsity pattern is known in advance, especially when complemented with on-the-fly filtering. When sparsity is enforced, only pre-existing blocks in the product $\mathbf{C}$ matrix are calculated. No new blocks are ever added to the $\mathbf{C}$ matrix.

The local multiplication is OpenMP parallelized. The rows of the $\mathbf{C}$ and $\mathbf{A}$ matrices are partitioned among the threads. Each thread then works independently on its subset of the process grid local rows. This technique ensures that each thread writes to memory local to the thread. In addition, no thread synchronization is needed in the multiplication kernel. The threading performance for a sample matrix is shown in Fig. 3.

When multiplying dense matrices, the blocks may be coalesced into larger, dense blocks to increase performance. A single block is formed from all the blocks used in each step of the local multiplication. Using larger, dense blocks leverages the benefit of vendor-provided BLAS implementations that tend to be optimized for large matrices. This is done whenever the input matrices are dense or almost dense, the resulting matrix has no sparsity constraint nor symmetry, and filtering is not used.

2.3. Implementation

The presented library has been publicly available since May of 2009 as part of the open source CP2K molecular simulation package [20]. In addition it can also be compiled and used as an independent library for broader use. Being co-developed as a core part of a widely-used package has ensured extensive real-world testing on a wide range of problems on many types of computers. This provides strong incentive to achieve the highest possible performance and deliver a robust library.

While the library was designed for matrix multiplication, its goal was to unify matrix data structures, both sparse and dense, used in the CP2K program. It therefore has a rich interface for: initializing matrices; adding, changing, and deleting blocks; iterating through all the matrix blocks; and more. It supports matrix addition, converting between different types and distributions, replicating them, and many others. The data types stored can be one of four types: real or complex numbers in single or double precision.

3. Results and Discussion

The focus of the DBCSR library is to perform fast multiplication of sparse matrices. The scalability and performance of the library is best demonstrated by its applications [19]. Nonetheless it must also maintain reasonable performance for dense or almost dense matrices. To assess the performance of the DBCSR library in meeting these goals we have performed several benchmarks that reflect the matrix data and multiplications found in production simulations using the CP2K program.

For all of the presented measurements, we measured the full time between the entry and the exit of a single call to the multiplication subroutine, which includes all setup and data preparation times, inclusive of mapping the virtual multiplication topology, Cannon's initial pre-alignment, possible data conversion, transposing, and duplicating matrices with symmetry, along with others. Since there is some variance in timings among several repetitions, the minimum observed time from several experiments is used. Unless otherwise specified we report the marketing performance of the matrix multiplication in (G)FLOP/s, which we have defined as the ratio between the theoretical number of floating point operations needed to perform the multiplication (i.e., $2MNK$ for the dense multiplication of a $M \times K$ and $K \times N$ matrix) and the time required to perform the calculation. We compare performance instead of time because it is

easier to compare it to the available processor performance. Reporting performance per core makes it easier to compare the strong scaling performance. We do not measure the actual performance, which is the ratio between the number of FLOPs actually issued and the time. What matters to the user is the time to solve a problem and not the actual implementation.

Dense Matrix Multiplication. For assessing the performance of dense matrix multiplication, we measured the marketing performance of multiplying vector-like matrices and adding the result to an existing matrix. Because this operation is common when using the default CP2K settings, it has to be ensured that its performance remains similar to the ScaLAPACK PDGEMM routine it was replacing. Two sizes were considered: a smaller test, multiplying a 5120×512 and 512×5120 matrix and adding it to an existing 5120×5120 matrix, and a larger test, multiplying a 20480×2048 and 2048×20480 matrix and adding it to an existing 20480×20480 matrix. Fig. 4 compares the performance of ScaLAPACK's PDGEMM as provided by the Cray libsci library, DBCSR dense mode in which blocks are coalesced, and DBCSR standard blocked mode. We used a block size of 32 for PDGEMM as it was found to be either the best or nearly the best for all the multiplications. The three tests were performed for both the smaller and larger matrix multiplication.

The tests were performed on palü, a Cray XE6 system installed at the time at CSCS, the Swiss National Supercomputing Centre. The compiler used is GNU gcc version 4.5.2 with the system-provided libsci library containing ScaLAPACK and BLAS implementations.

The DBCSR library provides a well-performing implementation of dense matrix multiplication. The performance is comparable to that of ScaLAPACK in the range of compared matrix sizes and core counts. Both libraries show a performance decrease with a higher number of cores, especially with the smaller matrix sizes. The performance of DBCSR drops by at most half when realistic blocking is used instead of the dense mode. The increased block count in the realistic matrices means that the indexing overhead is much greater, potentially impacting performance. However, subsequent testing indicates that indexing time is less than 10% of the total multiplication time. The bulk of the total time is consumed by calls to the optimized small matrix multiplication library. In addition the expected performance of small matrix multiplications of the 5×5, 5×13, and 13×13 matrices is worse than matrix multiplication of more machine-appropriate block sizes such as 16×16. However, subsequent experiments show that the difference is minimal. Thus most performance loss is due to less efficient multiplication of small matrix blocks compared to coalesced blocks.

Sparse Matrix Multiplication. For assessing the performance of multiplying sparse matrices several tests were performed. Multiplications $\mathbf{C} := \mathbf{C} + \mathbf{A} \times \mathbf{B}$ of 47104×47104 square matrices with several sparsities for all three matrices ranging from 20% to 99% (i.e., occupation from 80% down to 1%) were performed. Insufficient memory prevented denser matrices from being treated. The matrix

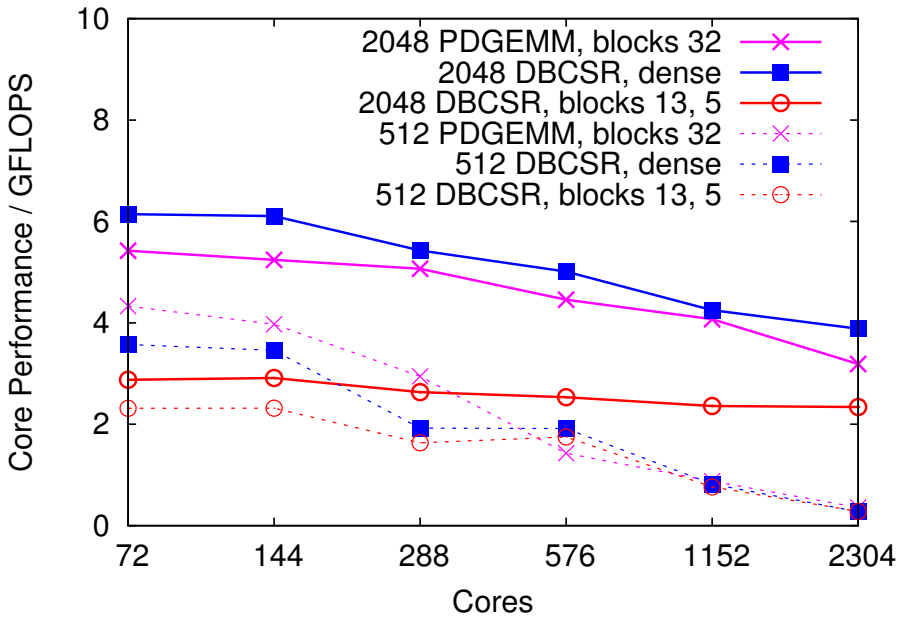

Figure 4: Performance comparison between ScaLAPACK PDGEMM and DBCSR dense and blocked mode dense matrix multiplication using different numbers of cores and two different matrix sizes. The performance measured is marketing performance, which we calculate as the floating point operations needed to multiply the matrices in a naïve serial implementation (i.e., $2MNK$ FLOPs for a product of a $M \times K$ and $K \times N$ matrix) and the time required for the entire operation, which includes all setup times. Performance curves for matrix multiplications of a 20480×2048 and 2048×20480 matrix are drawn as solid lines (labeled 2048) and performance curves for matrix multiplications of a 5120×512 and 512×5120 matrix are drawn as dashed lines (labeled 512). ScaLAPACK PDGEMM performance (labeled PDGEMM) is drawn in magenta color. DBCSR dense-mode matrix multiplication is drawn in blue (labeled DBCSR, dense) and DBCSR blocked-mode multiplication is drawn in red (labeled blocks, 13, 5).

was chosen to represent a system of 2048 water molecules, so it was blocked with realistic block sizes of 13 and 5. The sparsity pattern of the product matrix **C** was enforced. Again, this type of multiplication reflects many common operations in CP2K. These tests were performed on the same Cray XE6 system described earlier.

We report two performance figures for the sparse matrix multiplication. One is the actual multiplication performance, which counts the number of FLOPs used to multiply the small blocks. The time is measured in the same way as for the dense matrix multiplication, which includes all setup times. To see the advantage of using sparse matrix multiplication we then compare the actual performance to the marketing performance, which is the number of floating point operations that would be needed to multiply equivalently-sized dense matrices ($2 \cdot 47104^3$ in this case) divided by the time required for the actual multiplication. Obviously the marketing performance should be greater than the actual performance although, as we later show, this is not always the case for nearly dense matrices. The marketing performance serves as a measure of the time saved—or lost—by using sparse instead of dense matrices.

The multiplication operation described was performed using 576 and 2304 cores. The actual and marketing performance per core is plotted in Fig. 5. A reference of 6 GFLOP/s/core was taken as the peak performance that we have observed for dense matrix multiplication on the machine. It is drawn with a black line in the figure.

In Fig. 5 the crossing point between the marketing performance and the peak core performance is at a sparsity of about 0.3. (i.e., a density of about 70%) using either core count. For a good implementation of dense matrices multiplication the marketing performance equals 6 GFLOP/s/core peak performance. The sparse matrix multiplication has a lower marketing performance up to the 30% crossing point. It is therefore better to use dense matrix multiplication than sparse multiplication in this region. For matrices sparser than 30%, the marketing performance of DBCSR is higher than the marketing performance of dense matrix multiplication, so it is much more efficient to use DBCSR than the equivalent dense matrix multiplication. The threshold of switching to sparse multiplication is even lower if on-the-fly filtering can be used.

The scaling behavior of the same matrix multiplication on a wider range of cores is shown in Fig. 6.

The DBCSR library is well-suited for performing sparse matrix multiplication for a wide variety of sparsities. The performance is very good for the target application of CP2K in which many matrices have a relatively mid-range sparsity of about 90–99% (i.e., 1–10% occupation) [19]. As can be seen in Fig. 5, the actual performance of the sparse matrix multiplication remains above 1 GFLOP/s for sparsities up to about 90%. The actual performance then steeply drops for higher sparsities. However, the marketing performance continues to rise, especially when a greater number of cores is used.

Strong Scaling Performance. The strong scaling performance of the DBCSR code for applications dealing with realistic input data is shown in Fig. 7. In

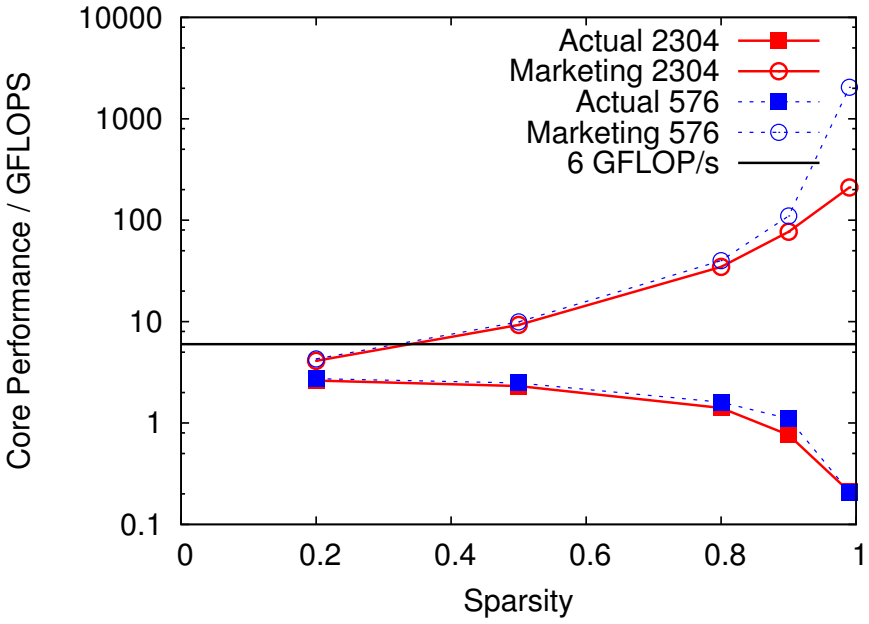

Figure 5: DBCSR sparse matrix multiplication performance for a variety of matrix sparsities. Actual performance (number of FLOPs used for multiplication divided by elapsed time) curves are drawn as lines with squares and marketing performance (number FLOPs that would be needed for dense multiplication of the same matrix divided by the elapsed time) are drawn as lines with circles. Performance on 2304 cores is drawn with red lines and performance on 576 cores is drawn with blue lines.

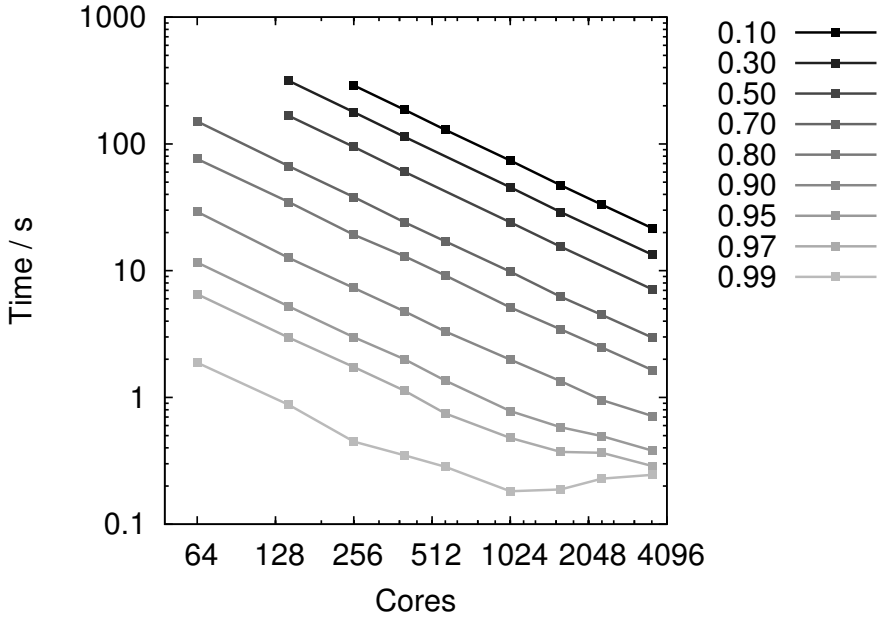

Figure 6: Scaling of DBCSR sparse matrix multiplication for a variety of matrix sparsities and core counts. The time needed for the multiplication of two 47104×47104 matrices with varying sparsities are drawn. Lighter shades indicate higher sparsity.

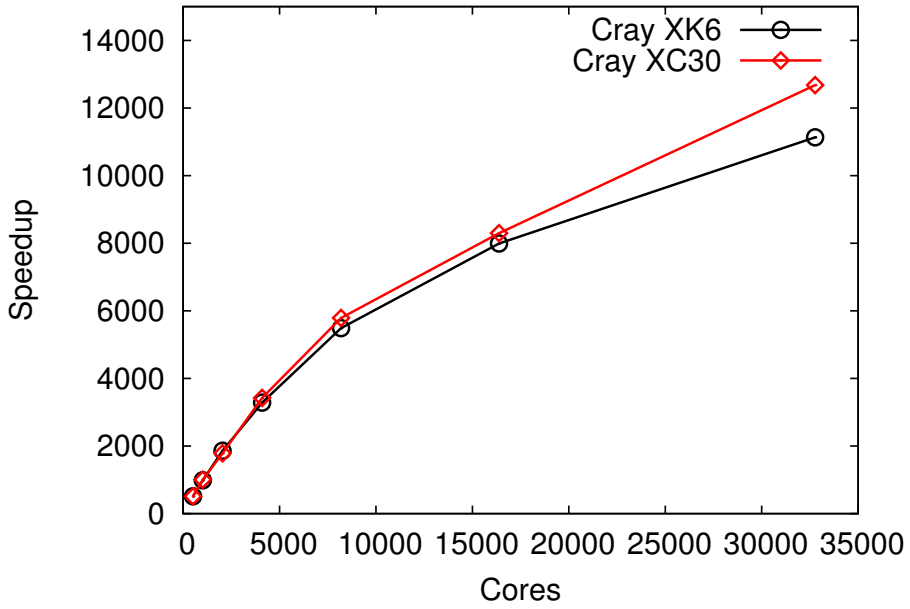

Figure 7: Observed speedup on two computers for a typical application with matrix sizes of 133214×133214. The two systems were a Cray XK6 (black curve, circles) and a Cray XC30 (red curve, diamonds). An ideal speedup is assumed for 512 or fewer cores.

this case, the electronic structure of an amorphous sample of an organic hole conducting material used in solar cells is computed. The sample contains 13846 atoms and 39560 electrons in a periodically repeated cubic unit cell with an edge of 53.84 Å. The corresponding square matrices have 133214 rows and columns. The benchmark requires 130 back-to-back matrix multiplications and has at most an 18% occupation of the matrices. Results are shown for two different architectures: a Cray XK6 with Interlagos CPUs and a Gemini network as well as a Cray XC30 with Sandy Bridge CPUs and an Aries network. On 512 cores, the smallest number due to memory reasons, the time spent in multiplication is 3351 s and 1555 s respectively. On 32768 cores the timings are 154 s and 63 s respectively. On the XC30 this implies that one multiplication takes less than 0.5 s on average. Good scalability required the use of threading and MPI rank reordering [21] on the XK6, while on the XC30 good performance was already observed before trying these techniques.

GPU Performance. In order to illustrate the potential of the GPU-enabled code we compare the GPU-enabled and the CPU-only version of the code in Fig. 8. The benchmark has been run on a system containing a Kepler K20 card and two Sandy Bridge E5-2620 CPUs (6 cores each, 2 GHz) using the threaded version of the code and all stacks processed by either the CPU or the GPU, exclusively. The matrix is blocked using submatrices of size 23×23, corresponding to the

basis of one H_2O molecule $(13 + 2 \cdot 5)$. Using regular parameter stacks as generated for the CPU, the GPU code reaches approximately 84 GFLOP/s while the CPU-only code peaks at 62 GFLOP/s using 12 cores. It can be observed that only a few CPU threads are necessary to reach full performance with the GPU code. Nevertheless, 84 GFLOP/s falls far from the peak performance of the K20 card, which is approximately 1 TFLOP/s.

To understand this discrepancy, it is instructive to analyze the basic $\mathbf{C}_{ij} := \mathbf{C}_{ij} + \mathbf{A}_{ik}\mathbf{B}_{kj}$ operation in terms of memory transfers and FLOPs. For simplicity we shall assume that $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are square matrices of dimension n. A total of $2n^3$ FLOPs are needed for the multiplication operation. If we assume that the above operation requires four transfers of $8n^2$ bytes, then a memory bandwidth of 200 Gbytes/s bounds the performance to $12.5n$ GFLOP/s. Further memory overhead (transferring stack data) and the actual time needed to perform computations further reduces the maximum performance, making the observed 84 GFLOP/s a reasonable result.

A potential strategy to further improve the performance is to sort the stack so that all updates to $\mathbf{C}_{ij}$ are performed consecutively, which avoids repeated loads and stores of $\mathbf{C}_{ij}$ and halves the memory bandwidth requirements. Employing sorted stacks, performance in excess of 121 GFLOP/s is observed. Despite these promising results, it remains an open challenge to exploit the GPU in the general case. In particular, multiplications of smaller matrix blocks are even more memory bound than the 23×23 blocks employed in this test and the overhead of sorting the stack becomes prohibitive for smaller blocks. Furthermore, while a hand-optimized CUDA kernel has been coded for this 23×23 case, a general framework for autotuning and autogeneration of a small matrix multiplication library for the GPU is still missing.

4. Application Performance: Linear Scaling Computational Cost

The main purpose of the DBCSR library is to enable calculations with a computational cost that grows linearly with system size. As an illustration in Table 1 we report the DBCSR performance for an important task in the application, namely the computation of $\mathbf{S}^{1/2}$ and $\mathbf{S}^{-1/2}$ from a given sparse square matrix $\mathbf{S}$ using Newton-Schultz iterations as discussed in [19], with a filtering threshold of 10^{-6}. The $\mathbf{S}$ matrix is obtained for a bulk liquid water geometry and has atomic block sizes of 5 and 13. As the system size increases, $\mathbf{S}$, $\mathbf{S}^{1/2}$, and $\mathbf{S}^{-1/2}$ become increasingly sparse. The latter two matrices are more occupied, having approximately 18600 non-zero elements per row, as can be seen from the occupation presented in Table 1. For this system, 35 matrix multiplications with varying sparsity pattern are needed to reach the desired convergence in the computation of $\mathbf{S}^{1/2}$ and $\mathbf{S}^{-1/2}$. The marketing performance remains relatively steady for all matrix sizes, and is in the range 1.0–1.5 GFLOP/s/core, decreasing slowly with system size and increasing sparsity. The time per multiplication varies widely from a very fast 0.09 s to more than 70 s for the matrices with a dimension exceeding 10^6. The normalized time, i.e., time per row, becomes approximately constant as soon as the occupation drops to about 20%. This

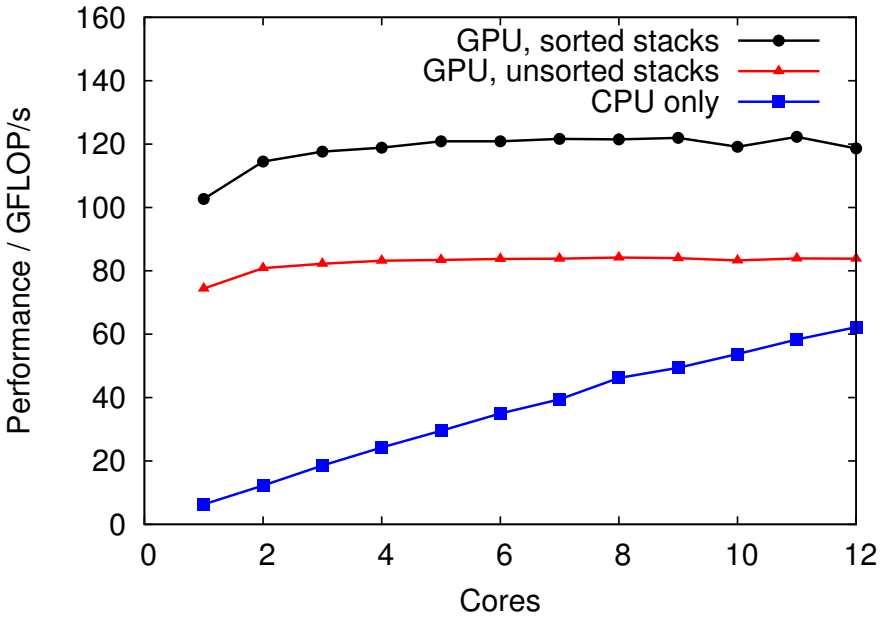

Figure 8: A comparison of the performance of GPU-enabled and CPU-only code for various numbers of threads (each running on a core). The lower line (blue squares) represents the CPU-only code while the two upper curves refer to the GPU-enabled code for two different strategies for building the parameter stacks: regular (red triangles) and sorted (black circles). For this benchmark using 23×23 blocks the best GPU performance is observed with sorted parameter stacks.

Table 1: Measured application performance for systems of increased size (see text for details) using 1024 MPI tasks with 2 threads each. Performance is reported as average time per multiplication, average time per multiplication normalized by the number of matrix rows, marketing performance, which assumes 2*dim^3 FLOPs, and actual performance, which is based on issued FLOPs as counted by the library. The normalized time remains effectively constant as soon as the occupation of the $\mathbf{S}^{-1/2}$ matrix becomes significantly different from one.

Dimension # waters	Dimension # rows	Occupation	Time s	Time/row μ·s	Marketing GFLOP/s/core	Actual GFLOP/s/core
256	5888	1.000	0.089	15.1	2	1.556
864	19872	0.813	0.775	39.0	10	1.485
2048	47104	0.395	2.063	43.8	49	1.306
4000	92000	0.203	4.267	46.4	178	1.360
6912	158976	0.117	8.125	51.1	483	1.232
10976	252448	0.074	12.178	48.2	1290	1.297
16384	376832	0.049	18.306	48.6	2855	1.173
23328	536544	0.034	27.173	50.6	5551	1.256
32000	736000	0.025	37.921	51.5	10267	1.235
42592	979616	0.019	50.472	51.5	18189	1.236
55296	1271808	0.014	72.864	57.3	27571	1.108

demonstrates the linear scaling capabilities of the DBCSR code. Finally, the marketing performance reaches 27 TFLOP/s/core for the largest system, implying that the same algorithm implemented with dense matrix algebra would require multi-petaflop performance to deliver the same time to solution.

5. Performance Limits

The performance of the DBCSR library allows for very large calculations on systems of scientific interest. Nevertheless, it is interesting to analyze and understand the limitations of the library so that further progress can be made. First, maximum FLOP rate is limited by the block size. Larger blocks imply a higher FLOP rate. High quality basis sets use bigger blocks and will thus yield higher FLOP rates but of course increase the total cost of simulations. Light elements, such as Hydrogen, have small basis sets, and hence reduce the overall FLOP rate. A higher FLOP rate can be obtained by coalescing related blocks, for example from nearby atoms in a molecule [11]. However, such coalesced blocks increase the occupation of the matrix and, despite the increased FLOP rate, might actually yield a longer runtime, the precise outcome depending on the system. Hierarchical storage formats alleviate this problem by grouping close blocks together without artificially increasing occupancy [13]. Furthermore, the book-keeping overhead needed to deal with the general sparsity of the matrices is non-negligible in practical calculations. Many operations, such as sorting the index, growing buffers for newly added blocks, merging data, and recursive block multiplication scale as $\mathcal{O}(N \log N)$, where N is the system size (and is linearly

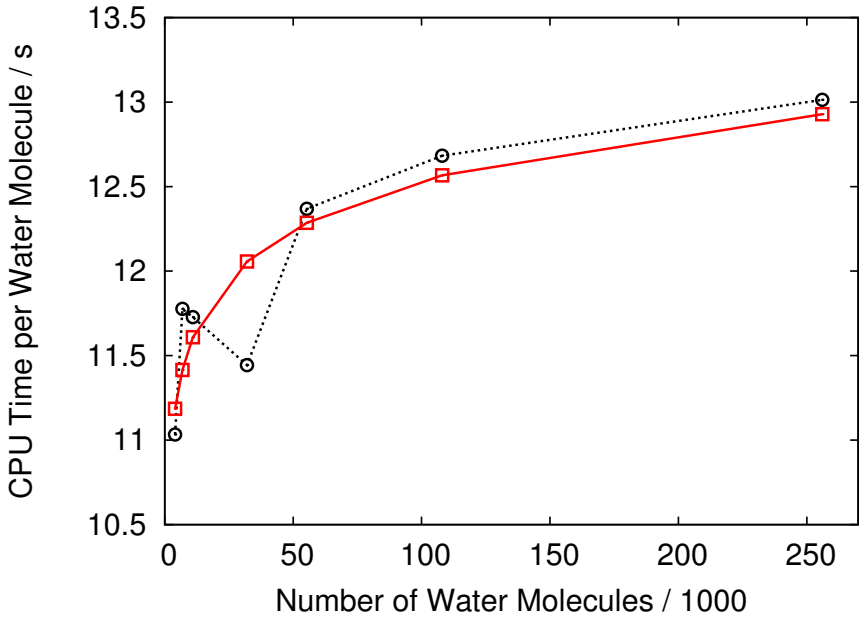

Figure 9: Shown is the CPU time per water molecule for systems in the range 4000–250000 water molecules on 2048 cores (1024 MPI × 2 OpenMP). The measured timings are shown with black circles on the dashed line while the solid red line is a fit of the data to the expected form of $y = A + B \ln x$.

related to all the matrix dimensions and, in the asymptotic case, the number of nonzero elements). Despite their small prefactor, they must ultimately dominate over the floating point operations, which are linear scaling—they scale as $\mathcal{O}(N)$. In Fig. 9, we demonstrate that this behavior can indeed be observed for test cases in which overhead is important. This particular test case is derived from a full self-consistent calculation of a semi-empirical model of water and is characterized by small blocks (molecular, 6×6) and high sparsity. It ran on 2048 cores (1024 MPI×2 OpenMP), and covers the range from 4000 to 250000 molecular blocks. The reported time amounts to 316 multiplications as needed for a full SCF cycle.

Presumably the most important limitation of the DBCSR library is related to our design choice of using the Cannon algorithm. Whereas the Cannon algorithm is optimal in the limiting dense case, it is not in the sparse limit. This limitation is most visible in weak scaling experiments in which the problem size, N, grows proportionally with the number of parallel tasks, P. In these experiments the amount of data and FLOPs local to a given process, $\mathcal{O}(N/P)$, is thus effectively constant since FLOPs and data both scale as $\mathcal{O}(N)$ for the linear scaling application of interest. However, the amount of data communicated is not constant because the volume of data moved in each of the $\mathcal{O}(\sqrt{P})$ steps of the Cannon algorithm is $\mathcal{O}(N/P)$ for a total volume of $\mathcal{O}(N/\sqrt{P})$. Hence, the

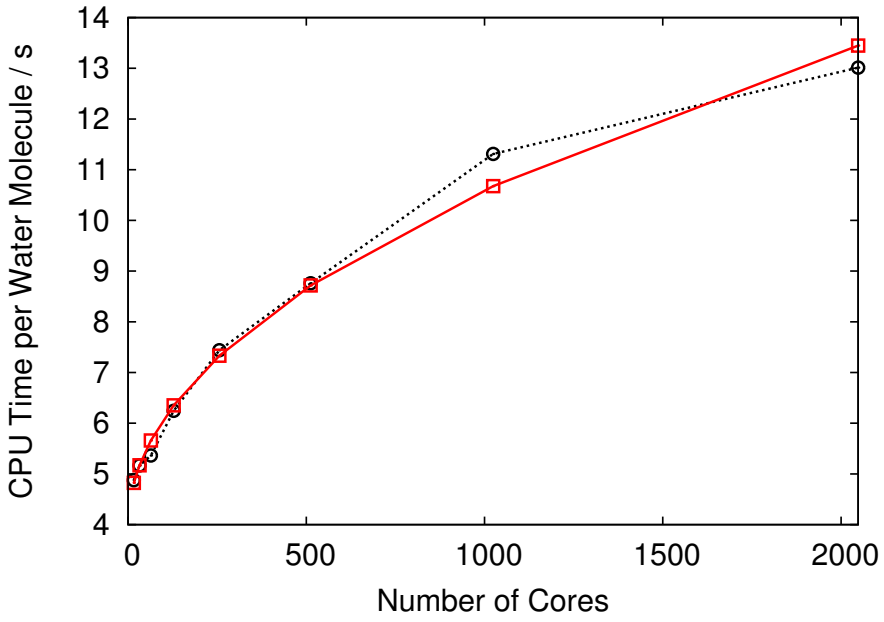

Figure 10: Shown is the CPU time per water molecule in a weak scaling experiment employing 128 water molecules per core, scaling from 16 to 2048 cores. The measured timings are shown with black circles on the dashed line while the solid red line is a fit of the data to the form of the expected simple model $y = A + B\sqrt{P}$.

FLOP-to-communication ratio decreases as $1/\sqrt{P}$ in these weak scaling experiments. This observation can also be shown theoretically [22] The bottleneck is more pronounced on networks where the effective bandwidth decreases with increasing P. Similar observations hold for indexing overhead, which also becomes more important as the number of multiplication steps increases. This limitation is also more pronounced for smaller blocks and its impact is illustrated in Fig. 10 for the same semi-empirical model of water discussed before. To overcome this limitation, it will be necessary to have a communication scheme that is different from the Cannon algorithm. The development of an algorithm for the multiplication of sparse matrices with a structurally-defined sparsity, that has the same performance as Cannon in the dense case but better behavior in the sparse limit is a challenging and ongoing research project.

6. Conclusions

Massively parallel linear scaling quantum simulations have become a practical tool to study systems of unprecedented size. Sparse matrix linear algebra is requisite for enabling such simulations. The DBCSR (Distributed Block Compressed Sparse Row) library described in this article is a well-performing library for scalable parallel sparse matrix-matrix multiplication with defined and

demonstrable scaling bounds. The library combines a number of techniques, including a modified version of Cannon's parallel matrix multiplication communication pattern, cache-oblivious recursive multiplication, multithreading, and element blocking. Finally an autotuning small matrix multiplication library provides fast multiplication for the basic element blocks. Filtering, both final and on-the-fly during matrix multiplication, increases sparsity as well as reducing the number of multiplication operations issued.

The library has good multiplication performance for a very wide range of matrix sizes, sparsities, and core counts. This wide range reflects the operations it must handle as a general-purpose matrix storage and multiplication library for the CP2K program.

Exploiting the sparsity pattern common to many matrices encountered in quantum chemical calculations holds promise to further improving the scaling behavior of parallel sparse matrix multiplication.

7. Acknowledgements

We are grateful to I. Bethune (EPCC) for MPI profiling and optimizations, P. Messmer (NVIDIA) for helpful discussions on CUDA optimizations, N. Stringfellow (CSCS) for CUDA kernel tuning, and Roberto Ansaloni (Cray) for the support with Cray systems and environment. The research leading to these results has received funding from the Swiss University Conference through the High Performance and High Productivity Computing (HP2C) Programme. JV acknowledges financial support by the European Union FP7 in the form of an ERC Starting Grant under contract No. 277910. Computer resources were provided by the Swiss National Supercomputer Centre (CSCS).

References

[1] J. Choi, J. J. Dongarra, R. Pozo, D. W. Walker, ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers, in: Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, 1992, 1992, pp. 120–127.

[2] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK: A portable linear algebra library for distributed memory computers–design issues and performance, Comput. Phys. Commun. 97 (1–2) (1996) 1–15. doi:10.1016/0010-4655(96)00017-3.

[3] R. Agarwal, S. Balle, F. Gustavson, M. Joshi, P. Palkar, A three-dimensional approach to parallel matrix multiplication, IBM J. Res. Dev. 39 (5) (1995) 575–582.

[4] J. Choi, A new parallel matrix multiplication algorithm on distributed-memory concurrent computers, Concurrency: Pract. Ex. 10 (8) (1998) 655–670.

[5] M. Krishnan, J. Nieplocha, SRUMMA: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems, in: Proceedings of the 18^{th} International Parallel and Distributed Processing Symposium, 2004, Vol. 1, IEEE, 2004, p. 70b. doi:10.1109/IPDPS.2004.1303000.

[6] S. Chatterjee, A. R. Lebeck, P. K. Patnala, M. Thottethodi, Recursive array layouts and fast matrix multiplication, IEEE T. Parall. Distr. 13 (11) (2002) 1105–1123.

[7] N. D. M. Hine, P. D. Haynes, A. A. Mostofi, M. C. Payne, Linear-scaling density-functional simulations of charged point defects in AlO using hierarchical sparse matrix algebra, J. Chem. Phys. 133 (2010) 114111.

[8] A. Buluç, J. R. Gilbert, Highly parallel sparse matrix-matrix multiplication, Tech. rep., UCSB (Jun. 2010).

[9] A. Buluç, J. R. Gilbert, Challenges and advances in parallel sparse matrix-matrix multiplication, in: Parallel Processing, 2008. ICPP'08. 37^{th} International Conference on, IEEE, Santa Barbara, CA, 2008, pp. 503–510.

[10] M. Challacombe, A general parallel sparse-blocked matrix multiply for linear scaling SCF theory, Comput. Phys. Commun. 128 (1–2) (2000) 93–107. doi:10.1016/S0010-4655(00)00074-6.

[11] C. Saravanan, Y. Shao, R. Baer, P. N. Ross, M. Head Gordon, Sparse matrix multiplications for linear scaling electronic structure calculations in an atom-centered basis set using multiatom blocks, J. Comput. Chem. 24 (5) (2003) 618–622. doi:10.1002/jcc.10224.

[12] D. R. Bowler, T. Miyazaki, M. J. Gillan, Parallel sparse matrix multiplication for linear scaling electronic structure calculations, Comput. Phys. Commun. 137 (2) (2001) 255–273. doi:10.1016/S0010-4655(01)00164-3.

[13] E. H. Rubensson, E. Rudberg, P. Sałek, A hierarchic sparse matrix data structure for large-scale Hartree-Fock/Kohn-Sham calculations, J. Comput. Chem. 28 (16) (2007) 2531–2537. doi:10.1002/jcc.20691.

[14] M. Challacombe, A simplified density matrix minimization for linear scaling self-consistent field theory, J. Chem. Phys. 110 (5) (1999) 2332–2342. doi:10.1063/1.477969.

[15] L. E. Cannon, A cellular computer to implement the Kalman filter algorithm., Tech. Rep. AD0692473, Montana State University Bozeman Engineering Research Labs (Jul. 1969).

[16] A. Buluc, J. R. Gilbert, Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments, SIAM Journal on Scientific Computing 34 (4) (2012) C170–C191.

[17] J. VandeVondele, M. Krack, F. Mohamed, M. Parrinello, T. Chassaing, J. Hutter, Quickstep: Fast and accurate density functional calculations using a mixed Gaussian and plane waves approach, Comput. Phys. Commun. 167 (2) (2005) 103–128.

[18] N. Bock, M. Challacombe, An optimized sparse approximate matrix multiply for matrices with decay, SIAM J. Sci. Comput. 35 (1) (2013) C72–C98.

[19] J. VandeVondele, U. Borštnik, J. Hutter, Linear scaling self-consistent field calculations with millions of atoms in the condensed phase, J. Chem. Theory Comput. 8 (10) (2012) 3565–3573. doi:10.1021/ct200897x.

[20] The CP2K program, URL: http://www.cp2k.org/.

[21] C. Pousa Ribeiro, J. Hutter, J. VandeVondele, Hierarchical mapping strategy for atomistic simulations on large-scale parallel machines, submitted for publication.

[22] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, S. Toledo, Communication optimal parallel multiplication of sparse random matrices, in: SPAA'13: Proceedings of the 25rd ACM Symposium on Parallelism in Algorithms and Architectures, 2013.